

Figure 1: Playing 500Hz sinusoid on audio output. Take signal source and audio sink block from the list of blocks. Set the data type of signal source to float. Set the frequency to 500Hz and sampling rate to 32k. Set the sampling rate in the audio sink block to 32k. Connect these blocks and execute the flow-graph.

1 GNU Radio

GNU Radio[1] is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic and commercial environments to support both wireless communications research and real-world radio systems.

A software radio is a radio system which performs the required signal processing in software instead of using dedicated integrated circuits in hardware. The benefit is that since software can be easily replaced in the radio system, the same hardware can be used to create many kinds of radios for many different transmission standards; thus, one software radio can be used for a variety of applications.

GNU Radio performs all the signal processing. You can use it to write applications to receive data out of digital streams or to push data into digital streams, which is then transmitted using hardware. GNU Radio has filters, channel codes, synchronization elements, equalizers, demodulators, vocoders, decoders, and many other elements (in the GNU Radio jargon, we call these elements blocks) which are typically found in radio systems. More importantly, it includes a method of connecting these blocks and then manages how data is passed from one block to another. Extending GNU Radio is also quite easy; if you find a specific block that is missing, you can quickly create and add it.

Since GNU Radio is software, it can only handle digital data. Usually, complex baseband samples are the input data type for receivers and the output data type for transmitters. Analog hardware is then used to shift the signal to the desired center frequency. That requirement aside, any data type can be passed from one block to another - be it bits, bytes, vectors, bursts or more complex data types.

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. GNU Radio Companion(GRC) is a Simulink-like graphical tool to design signal processing flow graphs.

GNU Radio supports several radio front-ends, either natively or through additional out-of-tree modules. We will be using Ettus Research USRP platform and RTL-SDR TV tuners.

1.1 First flow-graph

Flow graphs are graphs (as in graph theory) through which data flows. Many GNU Radio applications contain nothing other than a flow graph. The nodes of such a graph are called blocks, and the data flows along the edges.

Any actual signal processing is done in the blocks. Ideally, every block does exactly one job - this way GNU Radio stays modular and flexible. Blocks are usually written in C++ (might also be Python); writing new blocks is not very difficult.

In order to illuminate this diffuse topic a little, let's start with an example. Our aim is to generate a sinusoid and play it on audio output. Open GNU Radio companion by executing `gnuradio-companion` on a terminal. The completed flow-graph is shown in Figure 1.

The blocks are connected at ports. The first block has no input port, it produces samples. Such a block, with only output ports, is called a source. In an analog fashion, the final block, with no outputs,

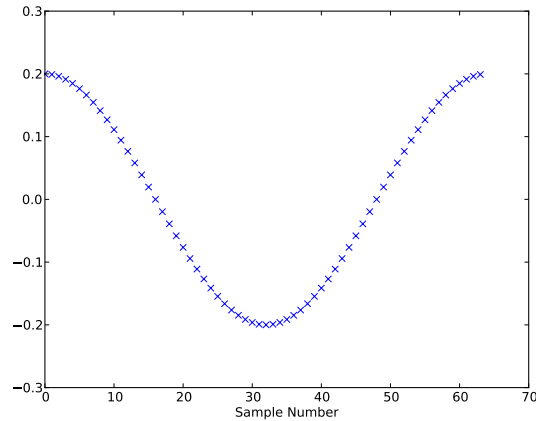


Figure 2: The first 64 samples of output from the signal source.

is called a sink.

So what's happening here? The signal source produces samples of 500Hz cosine sampled at 32kHz per second. The first 64 samples (which corresponds to a complete cycle of cosine in this case) is plotted in Figure 2. These samples are consumed by the multiply constant block, which multiplies each of the input with 0.5. Finally, the samples are passed to the audio output block which plays the incoming samples using computers audio card.

In general, we call whatever a block outputs an item. In Figure 1, one item was a float value representing one sample produced by the signal source. However, an item can be anything that can be represented digitally. The most common types of samples are real samples (as before), complex samples (the most common type in software defined radio), integer types, and vectors of these scalar types.

In Figure 1, apart from playing the samples on audio card, everything else is done in software. Since the audio card can only take specific sampling rates(32kHz, 44.1kHz, etc) the rate at which the samples are processed through this flow graph will be equal to the sampling rate of the audio card. As mentioned before GNU Radio applications are written in python programming language. When we run any flow-graph in GNU Radio companion, it is a python program that runs behind the scene. The python program corresponding to the flow-graph in Figure 1 is given in Listing 1. This program is auto-generated when we run a GNU Radio companion(GRC) flow graph.

Let us consider another example of computing the FFT of a signal source. The flow graph is given in Figure 3. In this example, the second block (stream to vector) produces one item(which is a vector of 1024 complex number) for every 1024 input items. So, the rate at which it produces items is 1024 times smaller then the rate at which it consumes items (the fact that it actually produces bytes at the same rate it consumes them is irrelevant here). Such a block is called a decimator, well, because it decimates the item rate. A block which outputs more items than it receives is called an interpolator. If it produces and consumes at the same rate, it's a sync block.

Let us ask the question what is the base sampling rate in Figure 3? Or how many complex numbers produced by the signal source, will be processed through this flow graph in a second? Well, we can't really answer this question. In this flow graph all that we are do, is to generate some numbers, do some math on the generated numbers, and save the output to a file. As long as there is no hardware clock present which fixes the rate, sampling rate is meaningless—only relative rates (i.e. input to output rates are important. The computer may handle the samples as fast as it wants. Note that this can cause the computer to lock up by allocating 100% of CPU cycles to your signal processing.

In order to avoid the computer getting busy doing all the math, it is recommended to use a throttle block when we run GNU Radio without any other hardware. The throttle block is tied to computer's hardware clock, and it throttles the number of items passing through it.

Also when we have hardware involving different sampling rates, we might have to use some up-sampling/down-sampling so that we meet the sampling rate requirement of all the hardwares.

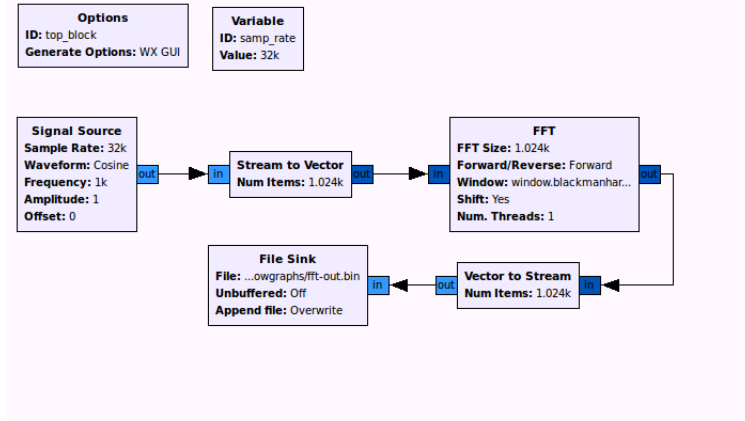


Figure 3: Computing FFT

2 IQ Modulation[4]

In RF communication we transmit our information in a small passband around a high frequency carrier. All the information carried in a real-valued passband signal is contained in a corresponding complex-valued baseband signal. This baseband signal is called the complex baseband representation, or complex envelope, of the passband signal. Any passband signal $s_p(t)$ can be written as

$$s_p(t) = \sqrt{2}s_i(t) \cos(2\pi f_c t) - \sqrt{2}s_q(t) \sin(2\pi f_c t) \quad (1)$$

where,

$$s(t) = s_i(t) + js_q(t) \quad (2)$$

is called the complex envelope of $s_p(t)$. In GNU Radio we work with this complex envelope. There are many hardware available which can be interfaced with GNU Radio to transmit the output from a flow-graph around any carrier that we may choose. Figure 4 compares the spectrum of complex envelope and passband signals. Figure 5 shows the process of IQ modulation. Figure 6 shows the process of IQ demodulation.

3 IQ Modulator Board

We have developed an IQ modulator board at Wadhwani Electronics Labs. This IQ modulator board features Linear Technology's LTC5598 modulator LTC6946-1 synthesizer and PIC18F4550 microcontroller. The block diagram of this board is given in Figure 7.

LTC 5598[2] is a high linearity direct quadraturemodulator, which works in the range from 5MHz to 1600MHz. It allows direct modulation of an RF signal using differential I and Q signals. The I/Q baseband inputs consist of voltage-to-current converters that in turn drive double-balanced mixers. The outputs of these mixers are summed and applied to a buffer, which converts the differential mixer signals to a 50Ω single-ended buffered RF output. The internal block diagram of LTC5598 is given in Figure 8.

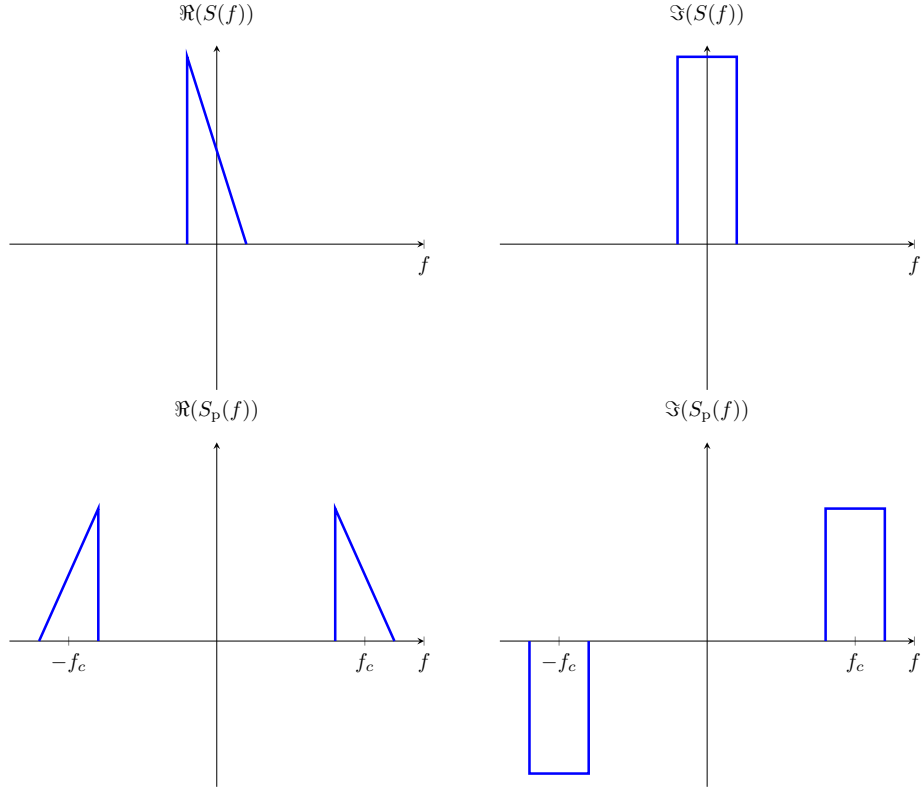


Figure 4: Spectrum of complex envelope and passband signals

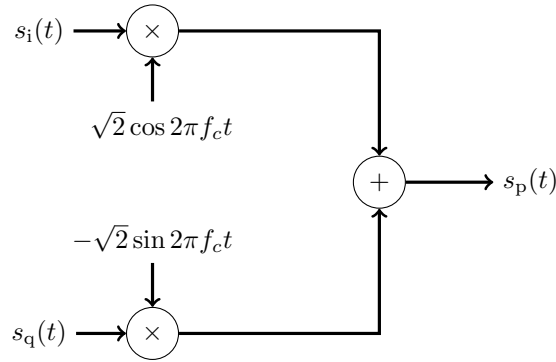


Figure 5: Up-conversion: Getting passband signal from complex envelope

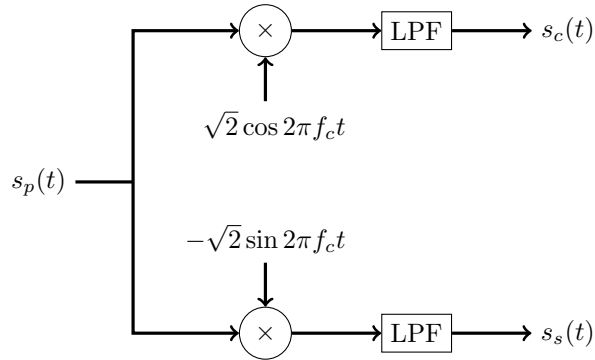


Figure 6: Down-conversion: Getting the complex envelope from passband

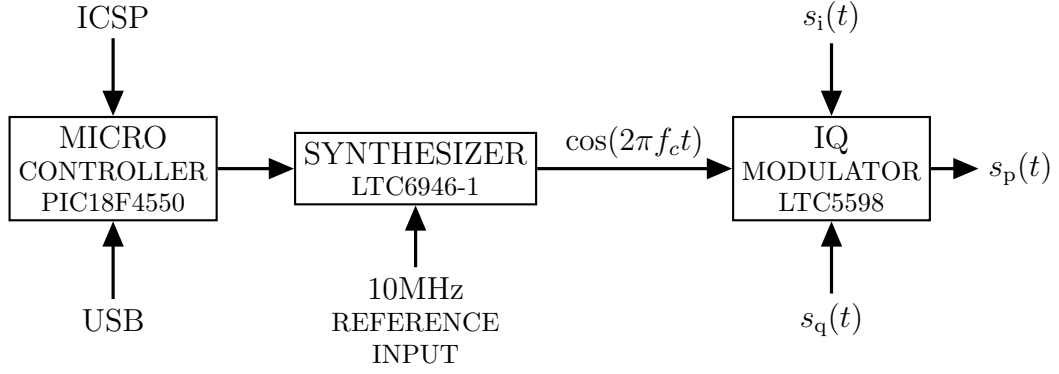


Figure 7: Block diagram of IQ modulator board

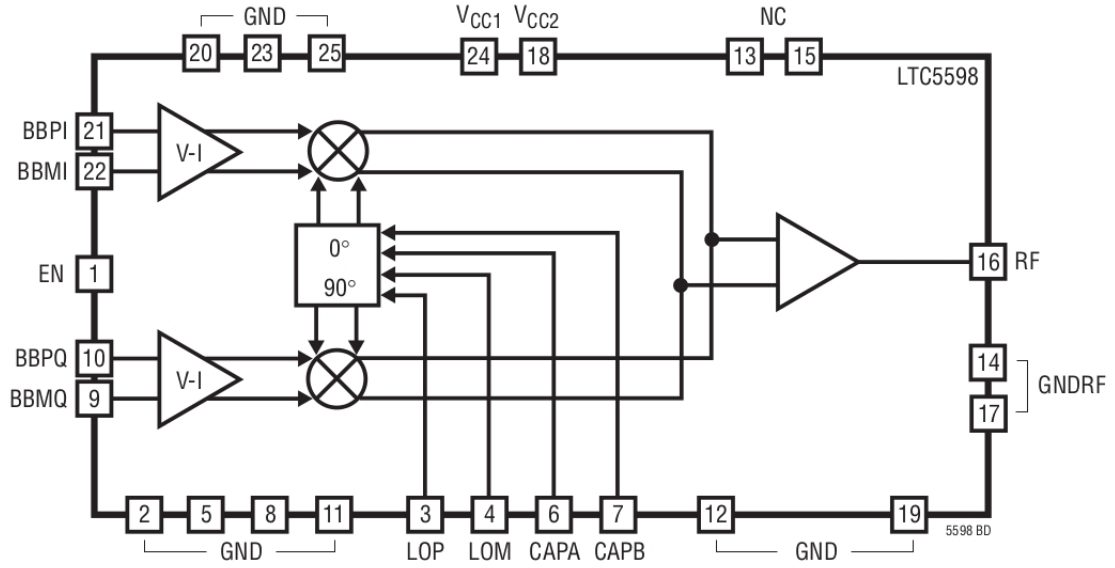


Figure 8: Internal block diagram of LTC5598 chip.

Appendices

```
1 #!/usr/bin/env python
2 #####
3 # Gnuradio Python Flow Graph
4 # Title: Audioplay1
5 # Generated: Sat Jun 28 18:50:14 2014
6 #####
7
8 from gnuradio import analog
9 from gnuradio import audio
10 from gnuradio import blocks
11 from gnuradio import eng_notation
12 from gnuradio import gr
13 from gnuradio.eng_option import eng_option
14 from gnuradio.filter import firdes
15 from grc-gnuradio import wxgui as grc-wxgui
16 from optparse import OptionParser
17 import wx
18
19 class audioplay1(grc-wxgui.top_block_gui):
20
21     def __init__(self):
22         grc-wxgui.top_block_gui.__init__(self, title="Audioplay1")
23         _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
24         self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))
25
26         #####
27         # Variables
28         #####
29         self.samp_rate = samp_rate = 32000
30
31         #####
32         # Blocks
33         #####
34         self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vff((0.5, ))
35         self.audio_sink_0 = audio.sink(samp_rate, "", True)
36         self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE,
37 500, 0.2, 0)
38
39         #####
40         # Connections
41         #####
42         self.connect((self.blocks_multiply_const_vxx_0, 0), (self.audio_sink_0, 0))
43         self.connect((self.analog_sig_source_x_0, 0), (self.blocks_multiply_const_vxx_0,
44 0))
45
46     def get_samp_rate(self):
47         return self.samp_rate
48
49     def set_samp_rate(self, samp_rate):
50         self.samp_rate = samp_rate
51         self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
52
53 if __name__ == '__main__':
54     import ctypes
55     import sys
56     if sys.platform.startswith('linux'):
57         try:
58             x11 = ctypes.cdll.LoadLibrary('libX11.so')
59             x11.XInitThreads()
60         except:
61             print "Warning: failed to XInitThreads()"
62     parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
63     (options, args) = parser.parse_args()
64     tb = audioplay1()
65     tb.Start(True)
66     tb.Wait()
```

Listing 1: The python program corresponding to the flow-graph in Figure 1

References

- [1] <http://gnuradio.org>
- [2] <http://www.linear.com/product/LTC5598>
- [3] <http://www.linear.com/product/LTC6946>
- [4] Upamanyu Madhov, “Fundamentals of Digital Communication”, CUP 2008.